AD-A273 233

USACERL Technical Report FF-93/12
September 1993

US Army Corps
of Engineers
Construction Engineering
Research Laboratories

Declarative Object Manipulation Environment (DOME): Alpha Version

by
R. Alan Whitehurst
John Pietrzak

The Declarative Object Manipulation Environment (DOME) is a programming language developed as part the Integrated Systems Language Environment (ISLE) that combines elements of object-oriented programming with knowledge-based and declarative programming facilities. The alpha version of this system, named ModLog (for "Modular Logic") was built as a declarative extension to the Army's ModSim language.

This report traces the evolution of declarative programming extensions from their beginnings in the Prolog programming language, through ModLog, to the first implementation of the DOME system. The integration of process-based simulation with DOME's declarative rule-based capabilities creates a powerful modeling paradigm that is superior to conventional imperative approaches.

93-29337
56pg

The contents of this report are not to be used for advertising, publication, or promotional purposes. Citation of trade names does not constitute an official endorsement or approval of the use of such commercial products. The findings of this report are not to be construed as an official Department of the Army position, unless so designated by other authorized documents.

DESTROY THIS REPORT WHEN IT IS NO LONGER NEEDED

DO NOT RETURN IT TO THE ORIGINATOR

USER EVALUATION OF REPORT

REFERENCE: USACERL Technical Report FF-93/12, *Declarative Object Manipulation Environment (DOME): Alpha Version*

Please take a few minutes to answer the questions below, tear out this sheet, and return it to USACERL. As user of this report, your customer comments will provide USACERL with information essential for improving future reports.

1. Does this report satisfy a need? (Comment on purpose, related project, or other area of interest for which report will be used.)

_____

_____

_____

2. How, specifically, is the report being used? (Information source, design data or procedure, management procedure, source of ideas, etc.)

_____

_____

3. Has the information in this report led to any quantitative savings as far as manhours/contract dollars saved, operating costs avoided, efficiencies achieved, etc.? If so, please elaborate.

_____

_____

4. What is your evaluation of this report in the following areas?

   a. Presentation:_____

   b. Completeness:_____

   c. Easy to Understand:_____

   d. Easy to Implement:_____

   e. Adequate Reference Material:_____

   f. Relates to Area of Interest:_____

   g. Did the report meet your expectations?_____

   h. Does the report raise unanswered questions?_____

i. General Comments. (Indicate what you think should be changed to make this report and future reports of this type more responsive to your needs, more usable, improve readability, etc.)

_____

_____

_____

_____

_____

5. If you would like to be contacted by the personnel who prepared this report to raise specific questions or discuss the topic, please fill in the following information

Name: _____

Telephone Number: _____

Organization Address: _____

_____

_____

6. Please mail the completed form to:

Department of the Army
CONSTRUCTION ENGINEERING RESEARCH LABORATORIES
ATTN: CECER-IMT
P.O. Box 9005
Champaign, IL 61826-9005

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	September 1993	Final

4. TITLE AND SUBTITLE
Declarative Object Manipulation Environment (DOME): Alpha Version

5. FUNDING NUMBERS
4A162784
AT41
SE-AU2 and SE-YC1

6. AUTHOR(S)
R. Alan Whitehurst and John Pietrzak

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
U.S. Army Construction Engineering Research Laboratories (USACERL)
P.O. Box 9005
Champaign, IL 61826-9005

8. PERFORMING ORGANIZATION REPORT NUMBER
TR-FF-93/12

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Office of the Chief of Engineers
ATTN: DAEN-ZCM
20 Massachusetts Avenue, NW
Washington, DC 20314-1000

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
Copies are available from the National Technical Information Service, 5285 Port Royal Road, Springfield, VA 22161.

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

The Declarative Object Manipulation Environment (DOME) is a programming language developed as part the Integrated Systems Language Environment (ISLE) that combines elements of object-oriented programming with knowledge-based and declarative programming facilities. The alpha version of this system, named ModLog (for "Modular Logic") was built as a declarative extension to the Army's ModSim language.

This report traces the evolution of declarative programming extensions from their beginnings in the Prolog programming language, through ModLog, to the first implementation of the DOME system. The integration of process-based simulation with DOME's declarative rule-based capabilities creates a powerful modeling paradigm that is superior to conventional imperative approaches.

14. SUBJECT TERMS
object-oriented programming
modeling
Declarative Object Manipulation Environment (DOME)
simulation
Modular Logic (ModLog)

15. NUMBER OF PAGES
54

16. PRICE CODE

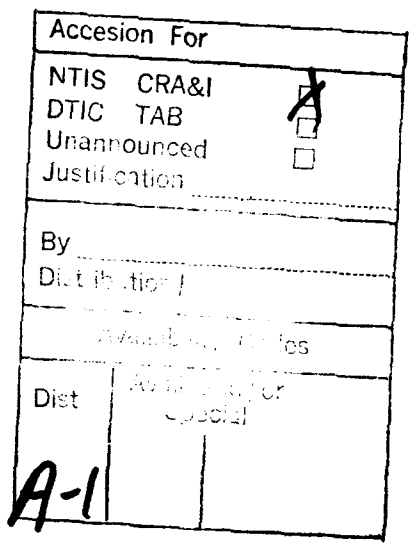
17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	SAR

FOREWORD

This study was conducted for the Office of the Chief of Engineers (OCE) under Project 4A162784AT41, "Military Facilities Engineering Technology"; Work Units SE-AU2, "Declarative Object Management Environment," and SE-YC1, "Functional Analysis Model Research Platform." The technical monitors were CAPT Robert Sinkler, ATSE-CDM-T, and Michael Shama, DAEN-ZCM.

This research was performed by the Facility Management Division (FF), Infrastructure Laboratory (FL), U.S. Army Construction Engineering Research Laboratories (USACERL). The USACERL principal investigator was Alan Whitehurst. Dr. Janet Spoonamore is Acting Chief, CECER-FF, and Dr. Michael J. O'Connor is Chief, CECER-FL. The USACERL technical editor was William J. Wolfe, Information Management Office.

LTC David Rehbein is Commander of USACERL, and Dr. L.R. Shaffer is Director.



CONTENTS

FIGURES

TABLE

DECLARATIVE OBJECT MANIPULATION ENVIRONMENT (DOME): ALPHA VERSION

1 INTRODUCTION

Background

Among those technologies identified by the Department of Defense as critical to national security are simulation and modeling (DOD 1990). Research in this area by the U.S. Army Construction Engineering Research Laboratories (USACERL) has included the development of combat engineering modeling and simulation technologies in support of the U.S. Army Engineer School (USAES), and research into simulation language technologies in support of the U.S. Army Training and Doctrine Command (TRADOC). In 1988, this project began to investigate the integration of modern software and hardware architectures to support large-scale simulation. This resulted in the U.S. Army ModSim, Version 1.0, the modular simulation language (Herring 1990). ModSim is a general-purpose fully object-oriented programming language that provides strong typing and modularity for programming in the large (Wirth 1984). ModSim also provides an object data type and integrates process-based discrete-event simulation with objects.

As a research test bed, a combat model was developed using ModSim to experiment with the application of software technologies and techniques. The Model-View-Controller framework of SMALLTALK (Krasner 1988) was used as a guide to designing both the global and local architecture of the model. From a study of existing models and knowledge of emerging software capabilities, two requirements became evident: (1) the need for declarative programming to model complex decision-making, and (2) the need for consistent object storage for model management. Work on these requirements resulted in the Persistent ModSim (Herring 1991, 1993), and ModLog (modular logic) declarative programming language (Whitehurst 1991, 1992).

ModSim is a procedural language based on Modula-2 with extensions to support object-oriented programming and process-based discrete-event simulation. To accommodate the ability to model complex decisionmaking by simulation objects, knowledge-based features were introduced to the ModSim language in the alpha version of the Declarative Object Manipulation Environment (DOME), which is a declarative extension to the Army's ModSim language.

DOME represents an evolution of the ModLog system specifically engineered for use as a tool within the Integrated Simulation Language Environment (ISLE). Tools in the ISLE environment interact with the persistent object repository, which stores information about the class hierarchy, the programs under development, and the results of program execution; all at the object-level of granularity (as opposed to the file, or function-level granularities that exist in most systems). To ensure that the alpha version of DOME remains distinct from later versions, it is referred to here as "ModLog."

Objective

The objective of this research was to investigate the integration of knowledge-base facilities into a procedural, object-oriented programming language to support the development of complex simulations. Simply put, the objective was to determine how simulation entities (as described by process-based simulation objects) could be extended to support complex nonprocedural reasoning.

Approach

A first-generation prototype was designed to extend the facilities of the object-oriented programming language ModSim to provide knowledge-based capabilities. The knowledge-based capability was required to include the ability to:

1. Represent knowledge
2. Reason about knowledge
3. Interface the results of the reasoning to the procedural program.

The declarative language was modeled on the Prolog declarative programming language. Prolog is a declarative language based on first-order predicate calculus and incorporating a backward-chaining (i.e., goal-oriented) evaluation scheme. Since the reasoning capability was to be added to simulation objects, a major consideration was the size and complexity of the resulting interpreter. Prolog is a very compact language and its implementations are generally smaller than alternative reasoning mechanisms, and Prolog incorporates its own reasoning strategy and knowledge-representation schemes. In the subsequent redesign, a tighter integration of the declarative language with its imperative host language was sought, to make the declarative interpreter an integral part of an object's capabilities.

Mode of Technology Transfer

U.S. Army ModSim, Version 1.0 has been distributed to Government agencies and their contractors along with a suite of example programs that aid in understanding the process-based simulation model of ModSim and general object-oriented programming, as well a graphical class-hierarchy browser and an editing/compilation management environment. It is anticipated that the completed program may be distributed cooperatively through several participating agencies: the Army Material Systems Analysis Agency, the Defense Logistics Agency, and through contractors on ARPA's next generation DIS War Breaker (Booze-Allen-Hamilton, Science Applications International Corporation, and The Applications Science Corporation).

2 RELATED RESEARCH

This study involved the integration of two separate programming paradigms (object-oriented procedural programming with logic-based declarative programming) within the context of computer simulation of complex models. A number of researchers have recognized the potential for integrating objects and logic programming and have reported on various approaches. Conery (1987), and Chen and Warren (1988) reported on approaches to capturing the semantics of object-orientedness in first-order logic, paving the way to the development of an integrated approach to object-oriented and deductive programming.

Most of the systems incorporate object-oriented concepts into logic or functional programming systems. SPOOL (Fukunaga and Hirose 1986), for instance, is an object-oriented language implemented on top of Prolog that represents methods as logic programs and regards a message to an object as the invocation of a goal. FOOPLog builds on Goguen and Meseguer's functional OBJ language to provide support for object-oriented programming. Other systems incorporate logical operations into the object-oriented paradigm e.g., Orient84/K (Ishikawa and Tokoro 1987) and KSL/Logic (Mamdouh and Cummins 1990) . These systems both represent objects as entities with two separate parts: a behavior part and a knowledge part. Their base object-oriented language is extended with pattern matching, unification, and backtracking to provide logic programming features. None of the systems cited so far have provided support for system simulation, but are rather examples of different approaches to integrating declarative and object-oriented programming facilities within a general-purpose programming language.

Two examples of simulation systems that integrate object-oriented and declarative features are BLOBS and ROSS-ART. BLOBS (Middleton and Zancouato 1986), a language specifically designed to support simulations, is built on the artificial intelligence language POP II. Mcfall and Klahr (1986) describe an extension to Rand Corporation's ROSS language to interface it with Inference Corporation's ART expert system language.

The DOME system differs from these mentioned in several important ways. First, it is specifically designed to support process-based simulation. Both the imperative language IMPORT and the structure of the declarative language DOME are optimized for the execution of discrete-event simulations. Second, the marriage of object-oriented programming with process-based simulation provides an optimum modeling environment. The paradigm of use associated with the integration of these two concepts involves decomposing a simulation into models of the real-world objects involved in the activity being simulated. Each object is described by the processes it can undertake. The addition of DOME to this scenario allows these models to be extended by capturing what each object knows, thereby separating out the control logic that would otherwise be embedded deep in the control structures of the imperative language and allowing for more complex and heuristic-based reasoning to drive the actions of the model objects. Finally, unlike the other systems, both IMPORT and DOME were specifically designed to work together in an integrated fashion.

3 PROLOG

Background

DOME is based on the logic programming language Prolog (short for "programming logic"), but has several advantages over other languages for representing and using knowledge. A short description of Prolog is useful in understanding both how DOME works and the design decisions made during development.

History of Prolog

Prolog was created over a decade ago at the Faculty of Sciences at Luminy in Marseilles, France. It was inspired by the resolution principle proposed by Robinson (1965), which suggested that a single powerful rule of inference could replace the multiple-rule systems of deduction proposed by logicians. The goal of the developers of Prolog was to create a general theorem prover that would also serve as the basis of a programming language that would enable a computer to simulate the thinking process by making deductions from sets of logical formulas. Such a language would allow programs to be developed in a declarative manner as specifications in mathematical logic (Colmerauer 1985).

Advantages of Using Prolog as the Basis for DOME

Prolog has several advantages over other types of languages for working with knowledge. Because it is declarative in nature, complex knowledge can be represented very simply. It has a built-in deductive search procedure, so the user need not worry about the manner in which to retrieve information from a knowledge base. And, as Prolog has existed for some time now, there is a large existing body of research built around it.

Declarative Programming

In Prolog, a programmer does not need to think about "control information" when writing a program. That is, all that needs to be written are a series of statements describing the problem to be solved. For example, the Prolog program shown in Figure 1 describes the membership relation between an item and a list of items. Since Prolog encodes lists in terms of a head (the first item of the list) and a tail (all the rest of the items in the list), there are two statements used to describe this relationship. First, if an item is the first element of the list, then it is a member of the list. Second, if an item is a member of the tail of the list, then the item is a member of a list. These statements correspond to the two lines of the Prolog program:

This program is declarative, as the user does not tell the computer exactly how to dereference an element of a list, or which statement to execute first. It can also be described in a procedural manner, if one knows how the goal resolution system works. Assuming the user asks the query "member(Value, List)" (where the user supplies the value and the list), a procedural reading of the member program would be: First, check whether Value is the first item in List; if it is, return true, otherwise recursively call

```
member(X,[X Tail]).
member(X,[Head Tail]) :- member(X,Tail).
```

Figure 1. Declarative Specification of Membership.

member() with Value and the tail of List. However, this procedural description can change, if the Value or the List is left "uninstantiated," where Prolog is supposed to find a value for it.

The use of control information could make this program harder to read. Figure 2 shows the member program written in a Pascal-like language. Note that, for simplicity, list elements are always integers in this example; Pascal does not support untyped variables, so a single type is used for each element. At first glance, it is obvious that the declarative specification shown in Figure 1 is much more compact than its procedural cousin. Membership can be specified in two relations. Membership is not defined in terms of "how," but rather in terms of "what." This is the essence of declarative programming (Kamiu 1988).

Search Algorithm

Prolog uses a backward-chaining rule system. That is, it starts with the goal it is attempting to prove, and attempts to show that the goal follows logically from some series of statements in the knowledge base. Thus, when Prolog is finished with a proof, it has worked backwards from the goal to the initial information provided by the user. With optimizations based on the fact that its programs are limited to Horn Clause logic (which is a simpler subset of full First Order Predicate logic), the search algorithm for Prolog is very efficient, and very fast.

Available Body of Research

During development, it was decided to keep DOME as close to Prolog as possible to capitalize on the extensive research done in the area of declarative programming using Prolog. This results in the ability to translate quite easily from existing Prolog code to DOME code. Unfortunately, this also means that DOME suffers from the same drawbacks as Prolog, and anyone serious about using either of these languages should fully comprehend their limitations. While solutions for many of these drawbacks are known, they either incur unacceptable performance penalties or require significant departures from Prolog's programming paradigm.

```
TYPE

   ELEM = INTEGER;

   ELEMLIST = POINTER TO ELEMLISTREC;

   ELEMLISTREC = RECORD
      head: ELEM;
      tail: ELEMLIST;
   END RECORD;

   PROCEDURE member(IN e:ELEM; l:ELEMLIST):BOOLEAN;
   VAR
      p: ELEMLIST;
   BEGIN
      p:=l;
      WHILE(p<>NIL)
         IF (e = p^.head)
            RETURN(TRUE);
         p:=p^.tail;
      END WHILE;
   END PROCEDURE;
```

Figure 2. Procedural Specification of Membership.

Disadvantages of Prolog

Unfortunately, Prolog does have a series of shortcomings. To be such an efficient knowledge management system, it makes some compromises. It is generally slower than a compiled language. Also, it is not a complete First Order Predicate Logic system, which limits to some extent the kinds of knowledge it can represent. Appendix A gives a more detailed description of the disadvantages of Prolog, along with some suggestions for solutions. Future versions of DOME are planned to include some of these suggestions.

Summary

Declarative programming can be a very useful way to simplify the encoding of knowledge into a program. Since Prolog can perform declarative programming very efficiently, it met most of the needs of this study. The only question was how to effectively integrate Prolog into an existing simulation language. The first attempt made to implement this integrated language was the ModSim/ModLog system, which is described in the next chapter.

4 MODLOG

Design Issues

ModLog was the first step taken to integrate the abilities of Prolog with a procedural simulation language. The major design issue was the level of integration between the two language systems. A very close integration could combine the syntax of the two languages more easily, and would make sharing data between the two systems much easier; on the other hand, a loosely integrated system would allow more flexibility in experimentation. For ModLog, the two languages were loosely integrated. This allowed the system to keep practically all the power of full Prolog; also, the underlying resolution system could be implemented separately from the procedural language, allowing work to progress on the two parts of the system somewhat independently. The disadvantages of this choice was a somewhat complex syntax for the resulting language system, and a cumbersome method of passing information between the two underlying languages.

Syntax of Clauses

The syntax of ModLog code is very similar to that of Prolog. It is based upon the Horn Clause logic system, and thus is normally written as a series of conditional statements. The form for each statement is:

IF item, item, ..., item THEN item.

where each item can be a constant, a variable, or a predicate expression. A predicate expression expresses a relationship between other items, and has the form of a function application. For example, to express an ordering relationship between two integers, one might write:

less_than_or_equal_to(10,20).

In this example, less_than_or_equal_to is the predicate expressing a relationship between the arguments, 10 and 20. An unconditional (always true) statement can drop the IF portion, as in the expression:

item.

Table 1 lists sample statements translated into ModLog syntax.

Modularization

In normal simulations, which can become quite large and complex, the logic that governs the decisions of a particular simulation object is usually embedded within and distributed across the entire code for the simulation. This makes it difficult to determine exactly what the object will do in a given set of circumstances. Altering the behavior of the object is even more complex, because code will have to be changed in a number of places for each behavioral change, and subtle interdependences may be disturbed, with unforeseen effects of the rest of the simulation. ModSim attempts to rectify this problem by combining object-oriented specification, modularization, and process-based discrete event simulation. In other words, objects are encapsulated by modules, and the actions of a ModSim object is encapsulated by the methods of that object. ModLog takes this a step further by separating the decisionmaking logic of the object from the actions of the object, and encapsulating these in a rule base.

Table 1

Sample ModLog Statements

English Statements	ModLog Statements
Horses have four legs.	IF horse(X) THEN has four legs(X).
Fish can swim.	IF fish(X) THEN can swim(X).
Spot is a dog.	dog(spot).
All dogs are mammals.	IF dog(X) THEN mammal(X).
Dog is man's best friend.	IF dog(X), master of(M,X) then best friend(M,X).

In addition to localizing the decisionmaking code into a single module, the object-oriented decomposition of the simulation into active simulation entities also facilitates a structuring of the knowledge. Instead of having a single knowledge base, each simulation object with reasoning capabilities has its own knowledge base, which captures its view of the world and personal strategies.

Information Exchange Between ModLog and ModSim

While ModLog is a general tool that can be used to solve any problem that can be stated in Horn Clause logic, its design assumed a certain paradigm of use, in which the rule-based systems constructed in ModLog would operate cooperatively with ModSim simulation objects. For ModLog to reason about simulation objects, a means was devised for the ModLog rulebase to access the state of ModSim objects that have been specifically introduced to the rulebase. This allowed rules to be written that reasoned directly about other objects. For instance, a Factory Foreman object could be granted access to view the states of all the Factory Worker objects. This would allow the knowledge base of the Factory Foreman to contain rules that reasoned about the most optimum distribution of workers in any given situation. For a class of objects to be visible to other object's knowledge-bases, the class must have inherited the visibility property by having the MLVObj (which is contained in the module MLV) as one of their ancestors, and must specifically override the BIND method of the MLVObj class to specify a representation of the class of objects in predicate form.

Efficiency Issues

Several additions to the design of ModLog were made for efficiency. Early in the design of the language, it was realized that the search procedure used a great deal of temporary storage during the course of a proof. Thus, a memory management system was built in to ModLog so that each proof would use a minimal amount of memory, which would be freed when no longer needed. The management design was not trivial, since attempting to keep track of when each individual expression is no longer needed would be far too expensive.

Also in the interest of memory efficiency, an iterative version of the search procedure was implemented. Prolog is normally implemented recursively; this means that each step in a proof will have to generate stack frames, jumps to subroutines, and perform other recursively oriented tasks in the machine it is implemented upon. ModLog uses an iterative system, so that each stack frame is declared in the high level language, and the search procedure uses a loop instead of subroutine jumps. Therefore, optimization can be performed both on the size of each frame, and on the execution of the loop.

An Example

The method of interaction with ModLog is just like Prolog; once a knowledge base has been established, ModLog can answer questions about the knowledge base. This is known as "querying the system." Figure 3 shows a knowledge base consisting of facts and rules about family relationships. When this rulebase is loaded into the interpreter, the knowledge base can accept questions such as "Is Homer the father of Bart?" In the syntax of ModSim and this particular rulebase, this question would be: father(homer, bart)?; to which ModSim replies: TRUE. A more interesting (and possible) question is: "Who is the father of Bart?" Stated in Mod-Sim, this is: father(Father, bart)? The capital letter at the start of Father makes Father a variable, which ModLog will attempt to instantiate with an appropriate value. In this case, ModLog replies: TRUE{Father=homer}.

Implementation of ModLog

There are several interesting facets to the implementation of ModLog. Those which will be considered in this chapter include the iterative search procedure, and the code to integrate ModLog with ModSim.

The Iterative Search Procedure

During the process of creating ModLog, the normal write-compile-test loop was performed many times. Since the usual method of implementing Prolog style theorem provers places the burden of actually performing recursion directly on the facilities of the programming language, the testing and debugging stage were time-consuming, and required an in-depth knowledge of the underlying language and how it handled procedure calls and recursion.

The algorithm presented in this paper performs all necessary recursive steps with a loop and a stack, reducing the amount of work needed to perform testing. Another benefit of using an iterative proof procedure is that, because it can place only the information important to the current goal of a proof onto the stack, time and space requirements can be reduced, allowing deeper recursive proofs to be attempted. Also, customized debugging procedures can be created, able to interrupt a proof at any point. Thus, the choice of an iterative proof procedure helped the implementation of ModLog in several ways.

Explicitly performing recursion using a loop and a stack is a simple, well understood process. However, this algorithm differs slightly from the usual recursive manner by which resolution is implemented in Prolog (van Emden 1984). The following paragraphs discuss the differences between the two systems.

```
IF parent(X,Y),member(Z,Y),male(X) THEN father(X,Z).
female(marje).
female(lisa).
female(baby).
IF parent(X1,X2),member(C1,X2),parent(C1,X3),member(C2,X3)
   THEN grandparent(X1,C2).
male(homer).
male(bart).
IF parent(X,Y),member(Z,Y),female(X) THEN mother(X,Z).
parent(homer,[bart,lisa,baby]).
parent(marje,[bart,lisa,baby]).
```

Figure 3. Family Relationships Knowledge Base.

Conventional Prolog proof strategies create a "resolution tree," where each particular unification (and substitution) between a goal of the theorem and a rule in the knowledge base is a branch of the tree. Each node in the tree needs to have a copy of all the goals not yet satisfied.

In a completed resolution tree, the leaves are those nodes that could not unify with any of the rules in the knowledge base. Leaves that have exhausted all the goals of the theorem are true proofs, while those that still have goals requiring unification are false. This resolution is normally optimized by only storing a single path through the tree at a time. The tree is traversed in a depth-first manner, and the nodes are stored as the arguments to recursive calls to the proof procedure.

Although van Emden's proof procedure performs unification in the same manner, it's method of storing the proof results are quite different. This system creates several "proof trees" during the course of a proof. A proof tree only has exactly one goal at each node, with children being all the subgoals necessary for the proof of that goal. Each leaf of this tree represents a goal that was proven without the use of any subgoals. A completed proof tree is equivalent to a single path through a resolution tree, with a leaf having all goals exhausted (a true proof). Proof trees for false proofs never get completed. As with resolution trees, each branch in a proof tree represents a unification. If a node does not unify, it is erased and the algorithm backtracks to the previous choice. (This is performed implicitly through backtracking in the recursive resolution system.)

The major difference between the two storage systems is that the structure-sharing system doesn't need to pass the entire set of goals to each node. If, at the current point in a proof, predicates Q, R, S, and T need to be proven, predicates R, S, and T will be copied into every node of a resolution tree during the proof of Q, since they need to be proven later on the same path. However, in a proof tree, the subgoals required to prove Q are children of the node containing Q; R, S, and T are in different branches. Thus, the storage size of nodes can be reduced greatly.

Other than their storage requirements, the algorithms are equivalent. Every time a resolution proof reaches a leaf, the structure-sharing system stops building a proof tree. If the leaf has exhausted all goals, the proof tree is complete. As the resolution tree backtracks, the proof tree backtracks and gets erased. Finally, the proof procedure is finished (in both systems) when it returns to the root of the tree and has no more choices to pursue.

Implementation of Iterative Resolution

ModLog's use of the structure-sharing proof system has been kept in modules that correspond to the different parts of van Emden's ABC algorithm (a depth-first tree search procedure that was adapted to his proof algorithm). To work with the other existing features of ModLog, some parts of the ABC algorithm have been significantly modified.

As noted before, this proof procedure performs a depth-first search of a proof tree. Beginning with the initial query, each goal to be proven is assigned a specific node, and resolved with the knowledge base. If a successful unification with a rule is achieved, the node is placed on the stack, and each subgoal (if any) is then given a node, and resolved against the knowledge base. Whenever a goal fails to resolve against the knowledge base, the prover backtracks to the last attempted goal, and tries another solution.

The modules in the ABC algorithm break this process down into three steps. Part A tries to find the next goal to resolve against the knowledge base; if there are none, the proof has been completed successfully. Part B attempts to unify the current goal with a rule within the knowledge base; success means searching for the next goal (part A), and failure means backtracking to the last goal (part C). Part

C removes the most recent goal from the top of the stack, and restores the state of the last goal being resolved with the knowledge base. These modules will be described in turn in the following sections.

Choosing the Next Goal To Attempt To Unify

The ModLog implementation of part A of the van Emden algorithm will look through the current proof tree for the next goal to unify (Figure 4). If none exist, the proof has been completed successfully. The state variable is a pointer into the stack, and the next goal variable is the goal it chooses as the next to unify against.

Choosing a Rule Within the Rule Base

The list shown in Figure 5 attempts to unify the current node with a rule in the knowledge base. The built-in features of ModLog are invoked here; primitive functions and arithmetic expressions are recognized and handled appropriately in the first half of the code, and the actual manipulation of the knowledge base occurs in the second half.

The Backtrack Algorithm

The backtracking algorithm works closely with the child procedure in Figure 5; that procedure places information about the current rule and substitution in each node on the stack, and the backtrack procedure retrieves that information (and restores the previous stack conditions) when a failure occurs (Figure 6).

```
ASK METHOD get_next_goal(
   INOUT state: FRAMEPTR;
   OUT next_goal: EXP
): BOOLEAN;

VAR goals: EXPLIST;
BEGIN
   goals := state^.nextchild;

   WHILE goals = NIL
      IF state^.father = NIL
         RETURN(FALSE);
      END IF;

      state^.father^.envlist := mkSublist(cpSubst(
         state^.father^.env, 0, TMPMEM),
         state^.father^.envlist, TMPMEM);

      compose(state^.father^.env, state^.env);
      state := state^.father;
      goals := state^.nextchild;
   END WHILE;

   next_goal := cpExp(goals^.head, 0, TMPMEM);
   applyToExplist(state^.env,next_goal^.args);
   RETURN(TRUE);
END METHOD;
```

Figure 4. ModLog Implementation of the van Emden Algorithm—Part A.

```
ASK METHOD child(
   IN goal: EXP;
   INOUT state: FRAMEPTR;
   IN id: INTEGER;
   IN cls_ptr: CLAUSE
): BOOLEAN;

VAR
   e1, e2: EXP;
   sigma: SUBST;
   new_frame: FRAMEPTR;
   rest: EXPLIST;

BEGIN
   (* Arithmetic Expressions *)
   IF goal^.etype = AREXP
      sigma := mkSubst(NIL, NIL, TMPMEM);
      IF NOT simArith(SELF, goal^.optr, goal^.args, sigma)
         RETURN(FALSE);
      END IF;
      make_new_frame(new_frame, sigma, state,
         NIL, NIL, id, goal);

      RETURN(TRUE);
   END IF;

   (* Primitives *)
   IF isPrim(goal^.optr)
      sigma := mkSubst(NIL, NIL, TMPMEM);
      applyToExplist(state^.env,goal^.args);
      IF NOT dispatchPrim(SELF,goal^.optr,goal^.args,
            state^.depth,state^.id,sigma, rest)
         RETURN(FALSE);
      END IF;
      make_new_frame(new_frame, sigma, state,
            NIL, NIL, id, goal);
      RETURN(TRUE);
   END IF;

   cls_ptr := get_candidate_clause(goal, cls_ptr);

   WHILE cls_ptr<>NIL
      e1 := cpExp(goal, 0, TMPMEM);
      e2 := cpExp(cls_ptr^.lhs, state^.id, TMPMEM);
      sigma := unify(e1, e2);

      IF sigma <> NIL
         make_new_frame(new_frame, sigma, state,
            cpExplist(cls_ptr^.rhs, state^.id, TMPMEM),
            cpClause(cls_ptr, 0, TMPMEM), id, goal);
         RETURN(TRUE);
      END IF;

      cls_ptr := get_candidate_clause(goal, cls_ptr);
   END WHILE;

   RETURN(FALSE);
END METHOD;
```

Figure 5. Unification of Current Node With Knowledge Base Rule.

```
ASK METHOD backtrack(
        INOUT state: FRAMEPTR;
        INOUT cls_ptr: CLAUSE;
        INOUT goal: EXP
    ): BOOLEAN;

    VAR tempframe: FRAMEPTR;

    BEGIN
       IF state^.reset_child <> NIL
          state^.env := state^.envlist^.head;
          state^.envlist := state^.envlist^.tail;

          tempframe := state;
          reset_frame(state);
          state := state^.father;

          WHILE state <> tempframe
             state^.env := state^.envlist^.head;
             state^.envlist := state^.envlist^.tail;
             state := state^.father;
          END WHILE;

          reset_frame(state);
       END IF;

       IF state^.father <> NIL
          cls_ptr := state^.nextdb;
          pop_frame(state);
          state^.nextchild := mkExplist(state^.reset_child^.head,
                state^.nextchild, TMPMEM);
          state^.reset_child := state^.reset_child^.tail;
          goal := cpExp(state^.nextchild^.head, 0, TMPMEM);
          applyToExplist(state^.env,goal^.args);
          RETURN(TRUE);
       END IF;

       RETURN(FALSE);
    END METHOD;
```

Figure 6. The Backtracking Algorithm.

The Loop and the Stack

The recursion has been replaced with a stack and a loop. The looping proof procedure used by ModLog (Figure 7) immediately returns the first true proof, if one is encountered, and exits; this is done for pragmatic purposes beyond the scope of this paper.

The stack holds all the information needed at any node of the tree. Figure 8 shows the ModSim declaration for the stack structure. Each new node is placed at the top of the stack, and any backtracking performed always erases the top element of the stack. This makes manipulation of the tree quite simple and efficient.

```
ASK METHOD prove (
   IN state: FRAMEPTR;
   IN id:    INTEGER;
   OUT result: SUBST
): BOOLEAN;

VAR
   goal: EXP;
   cls_ptr: CLAUSE;

BEGIN
   WHILE get_next_goal(state, goal) = TRUE
      cls_ptr := NIL;
      WHILE child(goal, state, id, cls_ptr) = FALSE
         IF backtrack(state, cls_ptr, goal) = FALSE
            RETURN(FALSE);
         END IF;
      END WHILE;
      id := id + 1;
   END WHILE;

   result := state^.env;
   RETURN(TRUE);
END METHOD;
```

Figure 7. ModLog Looping Proof Procedure.

Resolution Strategy

As already noted, several problems with Prolog affect its soundness and completeness. This version of the ModLog system uses standard Prolog strategies rather than one of the extensions. Therefore, ModLog implements standard SLD resolution, and uses the conventional depth-first search strategy. It does, however, include an occur check to prevent the problem of cyclic bindings and infinite trees.

Conclusions

One of the most important factors studied during the creation of ModLog was the amount to which a procedural language and a declarative language could be integrated. The question is not trivial; during development, many obstacles to joining the two languages were uncovered. However, the models implemented using ModLog have shown just how powerful is the ability to include a knowledge representation system in a simulation.

After successfully implementing ModLog, several ideas were proposed for extending the entire language system. These suggestions are being implemented in DOME, which will be the full implementation of Modlog. The major emphasis has been to more fully integrate the declarative and procedural languages; users of DOME should be able to work without using the "bind" system, and several other benefits will become available. The experience gained with respect to the problems of integration in ModLog were invaluable for the development of the new language. DOME is discussed in the next chapter.

```
FRAMEREC = RECORD
   mtype:    MEMTYPE;

   father:  FRAMEPTR; (* previous node in proof tree *)
   nextchild: EXPLIST; (* next children to check *)
   reset_child: EXPLIST; (* previous children checked *)
   env:    SUBST; (* list of substitutions *)
   envlist:  SUBLIST;    (* reset lists of substitutions *)

   depth: INTEGER; (* depth of the current branch *)
   id:  INTEGER;  (* id of current frame *)

   nextdb:  CLAUSE;    (* holds clause for backtracking *)

   prevframe: FRAMEPTR;    (* previous frame of stack *)
   nextframe: FRAMEPTR;    (* next frame of stack *)
END RECORD;
```

Figure 8. ModLog Stack Structure Declaration.

5 THE DECLARATIVE OBJECT MANIPULATION ENVIRONMENT

DOME is the most recent of a series of systems designed to ease the encoding and manipulation of knowledge within a simulation. It draws on the experience gained from Prolog and ModLog to perform this task effectively, but it extends these languages by adding several important features. First, as the acronym implies, DOME is meant to work in an object-oriented manner. Second, DOME is designed to function within the context of a persistent simulation language (Herring and Whitehurst 1991), and to exploit the advantages of having an object-oriented database management system to help manage its knowledge. DOME is also designed to implicitly encode knowledge directly from the procedural source code, which gives the interface between the procedural and declarative engines a seamless appearance to the user. Finally, DOME is integrated into the procedural language to a much greater extent than its predecessors.

Design Issues

The most important overall design issue was the close integration of DOME and IMPORT. (Many of the other issues had already been addressed in the implementation of ModLog.) This issue affected both the syntax of DOME, and the data types that it supports. It also added the requirement of supporting object-oriented features and persistence. The following sections generally describe the effect on the design of DOME of the seamless integration of DOME and IMPORT.

Syntax

The syntax of DOME was affected in that the user must now specify some information about how to transfer variables between IMPORT and DOME. Instead of inheriting the declarative capability from an abstract class (as was the case in ModLog through inheritance from the ModLogObj), a query to a DOME knowledge base now takes the form of a standard IMPORT method invocation. The method declaration must appear in the declaration module, with a boolean function signature. The name of the method that forms the query is taken as the name of the DOME predicate, and any arguments of the method become the parameters of the predicate. Results that bind new values to a parameter are actually stored in that variable. The intent of this design is to make the query system more intuitive to the user.

In ModSim/ModLog, data was exchanged between the procedural language and the declarative language only through the bind process; otherwise, both languages had completely independent data storage. With IMPORT and DOME, both share the same data. With respect to the syntax of the language, this allows the user to simply name the variables to be passed within the query to the knowledge base (as is done in a normal procedure invocation). However, the user must also mark each variable as "instantiated" or "uninstantiated," depending on whether DOME is to assign a value to the parameter. Essentially, DOME will treat instantiated variables as constants. Uninstantiated variables will be treated as logical variables, and no effort is made to add their values to the information being used in a proof. It is assumed within the course of a proof that each variable only contains one value at a time, so a variable that already contains a value cannot have a new value assigned to it.

Data Types

The most important change to DOME arising from the sharing of data with IMPORT is the difference in the data types supported. The new types supported will be arrays, booleans, pointers (although these were implicitly supported in ModLog with the bind system), and objects. As Prolog

generally performs only comparisons and assignment with data, adding each of these types to the proof procedure is simple.

A more complex change to the data type system of DOME is the change to C-style enumerated types, instead of Prolog-style symbols. In Prolog and ModLog, symbols are declared implicitly; any identifier with the correct syntax and in the correct context will automatically be treated as a new symbol. This is inadequate to accomplish the tight integration desired for IMPORT and DOME, because to share the results of symbolic computation, there must be an agreed-upon semantics for the symbols that are manipulated by DOME and then returned to IMPORT as the results of queries. IMPORT already supported the enumerated type, and as enumerated types are used for similar purposes in C as symbols are in Prolog, it was decided to use them in place of symbols. However, they have a superset of the capabilities of symbols; not only can one enumerated type be equal or unequal to another, they can also be greater or less than each other. In fact, the user can assign each enumerated type a specific value; thus, DOME's internal naming scheme for symbols will become slightly more complicated.

Object-Oriented Features

DOME will support object-oriented programming by incorporating knowledge directly into each object generated in IMPORT. Thus, the knowledge base will be distributed over the entire object hierarchy. Queries will be performed locally over the information available within each object that initiates a query. Inheritance will be supported generally by treating each clause within an object as a separate method, such that normal techniques of inheriting or overriding methods can be used.

Persistence

Another design factor for DOME is the issue of persistence. IMPORT works directly with an object-oriented database such that, at any point in a program, the state is saved within the database. However, due to the temporary nature of data within a proof, persistence is a more complex question in DOME. Certainly, a knowledge base should be persistent, as the information within it may be used over and over, but the temporary storage used within a proof is quite large, and normally ignored when the proof has finished. Also, most queries do not take long to execute, so that not much is lost if data during a proof is not saved. Thus, it seems logical that only part of the execution of a DOME program should be made persistent. Currently, that is the manner in which DOME has been implemented, but other considerations may change this in the future.

Implicit Data

One of the most fascinating aspects of the DOME project is the way knowledge will be created automatically from the structure of an object. Essentially, all of the data available within an object will be inserted into the knowledge base for that object. Thus, each object will be able to reason about its own state.

In ModLog, the BIND method allowed any object that had inherited the MLVObj (ModLog Visible Object) class to be bound to a knowledge object's knowledge base. This strategy allowed rules to be written that reasoned over the state of other objects. The motivation for this capability was the need to model cooperative and supervisory interaction between knowledge objects. For instance, a Factory-Foreman object might want to know the state of both the set of machines and the set of factory workers on duty, to make decisions about the optimum work assignments. Or two tank commanders might need to know what each other's recent orders were, to coordinate an attack. While the BIND granted the programmer the flexibility to make external objects visible to a knowledge base, and supported automatic updating of the state of those visible objects, there were several objections to this approach. First, the

BIND violated the principle of object-oriented encapsulation, in that external objects could see the state of a visible object without having to access a method of the object as an interface. Second, it granted too much insight into the operation of other objects. This second concern is really tied to modeling methodology. In the example of the FactoryForeman mentioned above, using the BIND strategy created an omniscient foreman, who instantly knew both the state of repair of the equipment and the activities of his workers. In the real factory environment, a foreman must rely upon reports from subordinates, instrument read-outs, and personal observation; in other words, his knowledge is imperfect. Given these objections to the BIND strategy, it was decided to omit the BIND construct from the initial version of DOME and to study the consequences of that design decision. The alternative to the BIND is to interface to other objects through method invocation (to return the their relevant state) and then pass that state as arguments to a query. However, each object's internal state will be available to the knowledge-base without the need to pass that state as an explicit argument to the queries.

The implication of this design feature is that programs will be able to reason about themselves to some degree. At the moment, all that is provided is the ability to reason about their own state; this is sufficient to allow an agent to choose a strategy that best conforms to current conditions within a model. In later versions of DOME, it is hoped that it will be possible for objects to also reason about their methods, in which they could actually adapt a strategy to fit a novel situation.

Future of DOME

Currently, DOME is still in its alpha release. In addition to finding and correcting problems with the current implementation, a number of extensions and explorations are planned to increase the functionality of the DOME system. These include:

- full first-order reasoning
- constraint satisfaction
- reflective reasoning.

The addition of the Model Elimination (ME) procedure to the SLD resolution strategy already employed by DOME, and an alternative search strategy, should allow DOME to be extended to support the specification of relationships in full First-Order Predicate Calculus (FOPC), while retaining the efficiency of the SLD procedure. (For a more complete discussion of the extension of Prolog-like systems based on Horn-clause logic to full FOPC, see Appendix A.) This is especially important to DOME because it may simplify some of the problems of modular inheritance of knowledge bases. Constraint satisfaction is an alternative inferencing procedure the deals with finding a set of values that are consistent with a given set of requirements (or constraints). IMPORT/DOME is intended to support engineering applications where constraint-based reasoning has been shown to be of great worth. Reflective reasoning is the ability of an object to reason about its state and behaviors, possibly with the goal of optimizing modifications.

6 SUMMARY

Object-oriented programming provides a powerful abstraction for representing real-world entities where encapsulation and inheritance facilitate development of modular and extensible systems, and logic programming provides powerful pattern matching and deduction capabilities. The integration of these two paradigms can provide considerable advantage over either object-oriented or logic programming alone when dealing with complex models.

Efforts to bring about this combination have uncovered some of the difficulties inherent in integrating two such different technologies. The assumptions that underlie Prolog about data types, language semantics, and even the resulting program itself, do not hold within DOME. The declarative programming aspects of DOME contribute to a system that exceeds the performance of other modeling languages.

The ModSim/ModLog language demonstrated that this integration was actually possible. The IMPORT/DOME language makes much more of the power of declarative programming accessible to the user. The application of a declarative language to a simulation package provides a unique advantage to any user attempting to represent knowledge within a model.

REFERENCES

Beer, Joachim, "The Occur-Check Problem Revisited," *The Journal of Logic Programming*, Vol 5 (1988), pp 243-261.

Boyer, R.S., and J.S. Moore, "The Sharing of Structure in Theorem-Proving Programs," *Machine Intelligence 7* (Edinburgh University Press, Edinburgh, Scotland, 1972), pp 375-398.

Chen, Weidong, and David Scott Warren, *Objects as Intentions* (SUNY at Stonly Brook, 1988).

Colmerauer, Alain, "Prolog and Infinite Trees," in K.L. Clark and S.A. Tarnlund, eds., *Logic Programming, APIC Studies in Data Processing*, Vol 16 (Academic Press, New York, 1982), pp 231-251.

Colmerauer, Alain, "Prolog in 10 Figures," *Communications of the ACM*, Vol 28, No. 12 (December 1985), pp 1296-1310.

Conery, J.S., Technical Report CIS-TR-87-09, *Object Oriented Programming With First Order Logic* (Dept. of Computer and Information Science, University of Oregon, 1987).

Department of Defense (DOD) Office of the Deputy Under Secretary for Acquisition, *DOD Critical Technologies Plan* (1990).

Fleisig,, S., D. Loveland, A.K. Smilley III, and D.L. Yarmush, "An Implementation of the Model Elimination Proof Procedure," *Journal of the ACM*, Vol 21, No. 1 (January 1974), pp 124-139.

Fukunaga, Koichi, and Shin Ichi Hirose, "An Experience With a Prolog-Based Object-Oriented Language," *OOPSLA '86*, pp 224-231 (ACM Press, New York, September 1986).

Genesereth, Mic' el R., and Nils J. Nilsson, *Logical Foundations of Artificial Intelligence* (Morgan Kaufmann Publishers, Inc., Palo Alto, CA, 1988).

Herring, Charles, and R. Alan Whitehurst, "Adding Persistence to an Object-Oriented Simulation Language," *Object-Oriented Simulation 1991* (Society for Computer Simulation, Simulation Councils, Inc., 1991), pp 72-80.

Ibrahim, Mamdouh H., and Fred A. Cummins, "Objects With Logic," *Journal of the ACM* (1990), pp 128-133.

Ishikawa, Y., and M. Tokoro. "A Concurrent Object-Oriented Knowledge Representation Language Orient84/k: Its Features and Implementation," *OOPSLA '87 Proceedings* (1987), pp 232-241.

Kamin, Samuel N., *Programming Languages: An Interpreter-Based Approach* (Addison-Wesley, 1988).

Korf, R.E., "Depth-First Iterative-Deepening: An Optimal Admissible Tree Search," *Artificial Intelligence*, Vol 27, No. 1 (September 1985), pp 97-109.

Loveland., D.W., "A Simplified Format for the Model Elimination Theorem-Proving Procedure," *Journal of the ACM*, Vol 16, No. 3 (July 1969), pp 349-363.

McFall, Michael E., and Philip Klahr, "Simulation With Rules and Objects, in J. Wilson, J. Henriksen, and S. Roberts, eds. *Proceedings of the 1986 Winter Simulation Conference* (SCS, 1986), pp 470-473.

Middleton, Sue, and Rob Zanconato. "Blobs: An Object-Oriented Language for Simulation and Reasoning," *AI Applied to Simulation; Proceedings of the European Conference* (Simulation Council, Inc. [SCS], San Diego, CA, 1986), pp 130-141.

Plaisted, David A, "The Occur-Check Problem in Prolog," in *Proceedings of the International Symposium on Logic Programming* (Institute of Electrical and Electronics Engineers (IEEE), March 1984), pp 272-280.

Robinson., J.A., "A Machine-Oriented Logic Based on the Resolution Principle," *Journal of the ACM*, Vol 12 (January 1965), pp 23-41.

Stickel., M.E., "A Prolog Technology Theorem Prover," *New Generation Computing*, Vol 2, No. 4 (1984), pp 371-383.

Stickel., M.E., "A Prolog Technology Theorem Prover: Implementation by an Extended Prolog Compiler, in G. Goos and J. Hartmanis, eds. *8th International Conference on Automated Deduction, Lecture Notes in Computer Science 230* (Springer-Verlag, New York, NY, 1985), pp 573-587.

Stickel, M.E., and W.M. Tyson. "An Analysis of Consecutively Bounded Depth-First Search With Applications in Automated Deduction," in *Proceedings of the Ninth International Joint Conference on Artificial Intelligence* (IJCAI, Los Angeles, CA, August 1985), pp 1073-1075.

van Emden, M.H., "An Interpretting Algorithm for Prolog Programs," *Implementations of Prolog* (Ellis Horwood, Ltd., Chichester, England, 1984).

Warren, David H.D.. *An Abstract Prolog Instruction Set*, Technical Note 309 (SRI International, AI Center, Menlo Park, CA, Oct 1983).

APPENDIX A: Deficiencies of Prolog

Introduction

This Appendix outlines the shortcomings of Prolog as applied to this domain of knowledge processing, and some of the methods that may exist to help overcome those shortcomings. It is quite likely that some of these methods will be implemented in the next generation of the DOME language.

Prolog

During development it was decided to keep DOME as close to Prolog as possible to capitalize on the extensive research done in the area of declarative programming using Prolog. This results in the ability to translate quite easily from existing Prolog code to DOME code. Unfortunately, this also means that DOME suffers from the same drawbacks as Prolog, and anyone serious about using either of these languages should fully comprehend their limitations. While solutions for many of these drawbacks are known, they either incur unacceptable performance penalties or require significant departures from Prolog's programming paradigm.

Problems

Some of the major advantages of Prolog over other programming languages are described in Chapter 2. However, Prolog does have some deficiencies, and these will be outlined below.

Although the principles governing Prolog's initial design were based upon mathematical logic, Prolog's mathematical heritage, originally viewed as an asset, proved increasingly burdensome during implementation. Finally, the developers chose to adopt more practical constraints and the resulting language, while providing a theoretical model of some importance, deviated from the logical foundation that motivated its development. Still, Prolog is of great interest to the theorem-proving community because it is extremely fast and efficient when compared to conventional theorem-proving environments. It is of interest to the AI communities because it supports a declarative problem formulation and a deduction strategy that can be used to model complex human-like reasoning. Although Prolog traces its roots to mathematical logic and Robinson's resolution principle, it suffers from three main logical deficiencies that prevent it from qualifying as a general-purpose theorem proving environment, and that can cause logical soundness problems when Prolog is used naively. These deficiencies are:

1. The type of resolution employed by Prolog systems is not complete for full First-Order Predicate Calculus (FOPC).

2. The omission of the "occur check" in the unification algorithm of most Prolog implementations renders Prolog logically unsound.

3. The search strategy employed by Prolog is incomplete.

The remainder of this appendix explores the implementation decisions that caused Prolog to fall short of being a general-purpose theorem proving environment, and to suggest some possible modifications to Prolog that would rectify these deficiencies, including: (1) an overview of Prolog's deductive mechanism, an explanation of why this mechanism is unsatisfactory for general theorem proving, and a proposal for an extension to Prolog's deductive strategy to correct the deficiencies; (2) a discussion of a

failing in the implementation of the unification algorithm used by most Prolog interpreters and includes several possible alternatives to correct this problem; (3) an explanation of why Prolog's depth-first search strategy is incomplete and includes a discussion of ways that the search strategy might be augmented to correct this deficiency; and (4) an analysis of the behavior of conventional Prolog in contrast to the behavior of an extended Prolog in dealing with a more complex proof.

Prolog's Resolution Strategy

SLD Resolution

Prolog programs consist of logical clauses from the subset of FOPC called Horn logic. A Horn clause is a universal clause that contains at most one positive literal. The inference procedure used by Prolog is a form of input resolution known as SLD resolution (Selection/Linear/Definite resolution).

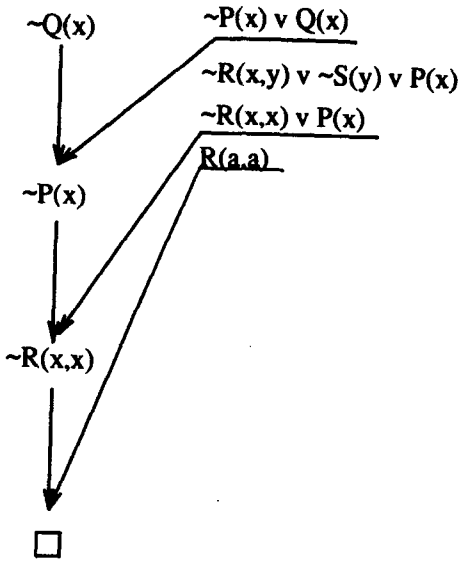


Figure A1. Sample SLD Derivation.

"Selection" refers to the fact that a selection function is used to guide the selection of literals upon which to resolve. "Linear" refers to the shape of the derivation. In a linear derivation, each derived clause is the result of resolving the most recently derived clause with a clause from the input set. Finally, "Definite" refers to the fact that, with the exception of the "goal" clause, all the input clauses in the derivation will be definite clauses. Definite clauses are the subset of Horn clauses that have exactly one positive literal. A sample SLD derivation of the empty clause from the unsatisfiable set $\{P(x) \supset Q(x), R(x, y) \wedge S(y) \supset P(x), R(x, x) \supset P(x), R(a, a), \neg Q(x)\}$ is shown in Figure A1. Note the shape of this proof there is a linear path from the goal to the empty clause. Each of the clauses in the input set that are of the form $L_1 \wedge L_2 \wedge L_3 \wedge \cdots \wedge L_n \supset G$ is a definite clause, because each of the literals on the left of the $\supset$ will be negated and disjoined when the clause is converted to universal form, leaving the single positive literal G. The goal (i.e., the top of the derivation tree) must always be a clause consisting of exclusively negative literals. Figure A2 represents the search tree for the derivation shown in Figure A1. The only point in which the left-most selection function comes into play during the construction of this derivation is when selecting that literal to resolve in the clause $\neg R(x, y) \vee \neg S(y)$. Unfortunately, the selection function makes the "wrong" choice in this case, for if the literal $\neg S(y)$ had been chosen instead of $\neg R(x,y)$,

the unprofitable branch of the search tree would have been pruned one node earlier. The selection function is only used when selecting literals from clauses it has no impact on the choice of which clause to use when more than one applies. The "or" branch under ¬P(x) in Figure A2 is representative of this situation. Since two separate clauses have literals that could resolve with ¬P(x), a decision must be made about which to attempt first. Prolog has adopted the convention that the clauses will be attempted in the order that they appear in the input set. (Prolog uses a "leftmost" selection function, which means that prolog will attempt to establish the validity of the literals in its goal clause in a left-to-right order.) Therefore, Prolog would generate the full search tree on its way to discovering the empty clause.

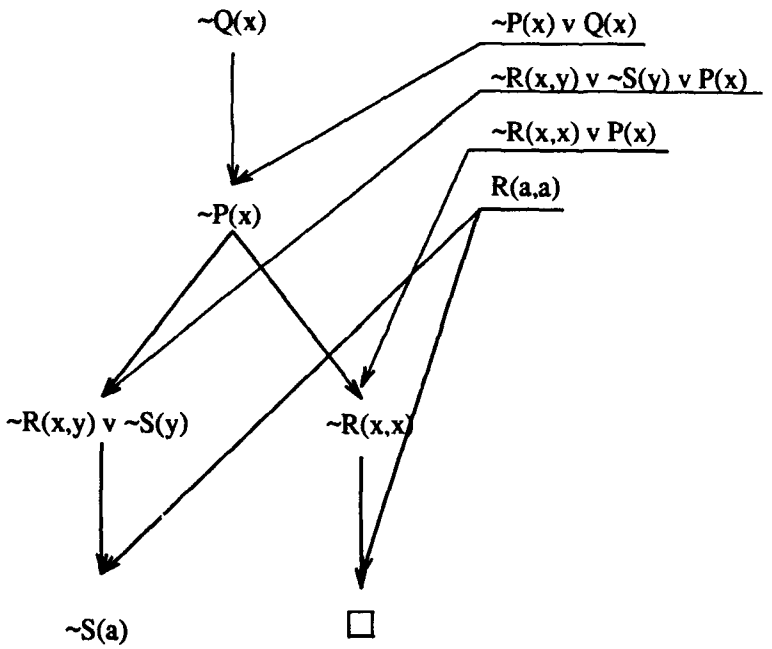


Figure A2. Search Tree for SLD Derivation.

SLD resolution is a form of input resolution, and like input resolution is incomplete for FOPC; therefore, if Prolog is to be extended so that it is complete for the full FOPC domain, a resolution procedure must be found to replace SLD resolution that is refutationally complete for FOPC while preserving those characteristics of SLD resolution on Horn clauses that allow Prolog to achieve its computational efficiency.

Model Elimination

It would be possible to extend Prolog's inference procedure so that it is complete for full FOPC by substituting any of the resolution strategies that are complete for FOPC for the incomplete SLD resolution used by Prolog. Examples of such strategies include Linear resolution and Set-of-Support resolution. Of course, it would be meaningless to extend Prolog's inference procedure in an attempt to make it complete for FOPC if the the restriction were retained that all clauses must be Horn clauses; therefore, the syntax of Prolog must also be expanded to allow the specification of clauses in FOPC. To allow the expression of full FOPC within the framework of Prolog's search strategy, one has only to introduce a true negation operator (¬) and allow this operator to appear anywhere in the input set.

In choosing a complete inference procedure to replace SLD resolution, note that the ability to execute Prolog efficiently is derived from several characteristics associated with input resolution strategies, such as:

1. As Prolog searches for a derivation, there is never more that a single derived clause at any given time.

2. As the derivation is constructed, any given variable can assume only a single value binding.

3. Since one of the resolvants is always a member of the input set, a Prolog program may be compiled into specialized resolution procedures that accept the derived clause as an argument and perform unification and resolution that is unique to the input clause.

Prolog's representation of derived clauses is an extension of structure sharing (Boyer and Moore 1972), in which only a single derived clause exists at a time and variables do not have multiple values; the ability to make use of this representation is a direct consequence of the decision to use a resolution procedure based upon input resolution (Stickel 1985). Therefore, if the advantage of Prolog's execution speed is to be retained, a complete inference procedure must be employed that is a form of input resolution that preserves the linear shape of the derivation.

$$\frac{\begin{array}{l} Q \supset P \\ Q \end{array}}{P}$$

Figure A3. ME Extension Operation.

Fortunately, such resolution strategies do exist. One inference system that is both refutationally complete for the full FOPC and still based upon input resolution is referred to as the model elimination (ME) procedure (Fleisig et al. 1974; Loveland 1969). ME consists of two basic operations, the extension operation and the reduction operation. The extension operation is illustrated in Figure A3. Informally stated, if in the course of attempting to prove the current goal P, it is found that Q implies P and Q is true, then P is true. The extension operation corresponds exactly to Prolog's normal inference procedure.

$$\frac{\begin{array}{l} Q \supset P \\ \neg P \end{array}}{P \supset Q}$$

Figure A4. ME Reduction Operation.

The ME reduction operation, which is illustrated in Figure A4, is a form of reasoning by contradiction. It states that if the current goal matches the complement of one of its ancestor goals, then apply the matching substitution and treat the current goal as if it were solved. In other words, if in the course of attempting to prove P, P follows from Q, and Q follows from ¬P, then one can conclude P.

These two operations together comprise an inference system that is refutationally complete for full FOPC. Therefore, to extend Prolog's inference system so that it is complete for FOPC requires only the addition of one additional inference operation¬ the ME reduction operation.

$$\left\{\begin{array}{c} Q(y) \wedge R(x) \wedge S(x,y) \supset P(x), \\ Q(x) \supset R(x), \neg P(x) \supset S(x,y), \\ Q(a), \neg P(x) \end{array}\right\}$$

Figure A5. Sample Set ME1.

As an example, consider the following unsatisfiable set ME1 shown in Figure A.5. Figure A6 shows this set rewritten for an extended Prolog, with the goal being to show P(x). The derivation representing the successful construction of a proof in an extended inference framework is shown in Figure A7. The last step of this proof involves the ME reduction operation, where P(x) and ¬P(a) are unified to complete the proof and yield the answer "a".

The program in Figure A6 could not be presented to a conventional Prolog interpreter because of the absence of the ¬ operator in Prolog2. Overlooking this fact for the moment, and assuming a Prolog that has a ¬ operator[1] that succeeds according to Prolog's conventional inference strategy (i.e., only when the negated fact is present as an assertion in the input set), it is obvious that Prolog's conventional inference strategy would still be insufficient to complete this proof because there is no fact establishing ¬P(a) in the knowledge base.

$$\begin{array}{rcl} P(x) & \vdash & Q(y), R(x), S(x,y) \\ R(x) & \vdash & Q(x) \\ S(x,y) & \vdash & \neg P(x) \\ Q(a) & \vdash & \\ & ?- & P(x) \end{array}$$

Figure A6. Prolog Equivalent of Sample Set ME1.

Disadvantages

Although the adoption of this inference system would allow the implementation of an extension of Prolog that was complete for FOPC and still reasonably efficient, it is not without disadvantages. These disadvantages may be grouped into two major categories:

1. The implementation of ME reduction within the efficient framework of Prolog requires that, for each assertion of N literals, N-1 additional assertions be made that are contrapositives of the original assertion such that each of the N literals is the head of one of the new assertions

2. Since the logical domain is full FOPC (not Horn Logic), a definite answer to a given query is no longer assured. This is reflected by the fact that if an indefinite answer is to be returned, the set of clauses must contain the negation of the goal clause in addition to the goal clause for the proof to succeed.

---

[1]Many Prolog implementations do have a not function, but this stands for negation as failure and has a different semantics than the true logical negation of FOPC.

The "Occur Check" Problem

Prolog's computational paradigm is based upon resolution. For efficiency, most Prolog implementations use a unification algorithm without an occur check. The occur check is used to determine whether or not a variable occurs within the term with which it is being unified. In the absence of the occur check, unification is unsound because it allows expressions such as P (x) and P (f(x)) to unify even though there exists no substitution for x that would make the two expressions equivalent (Genesereth and Nilsson 1988).

An unsound unification algorithm makes it possible to write formally correct Prolog programs that derive nonsensical results. The most noted example was created by Plaisted, in which a three clause Prolog program is constructed that proves 3 < 2 (Plaisted 1984). Figure A8 shows the set of clauses from Plaisted's example in a format that could be submitted to a Prolog interpreter. Let lt stand for < and S(x) be the successor function on the domain of integers. Given such a set of clauses and the query ? lt(3, 2), Prolog would respond "**yes", signifying that it was able to find a proof that three was indeed less than two.

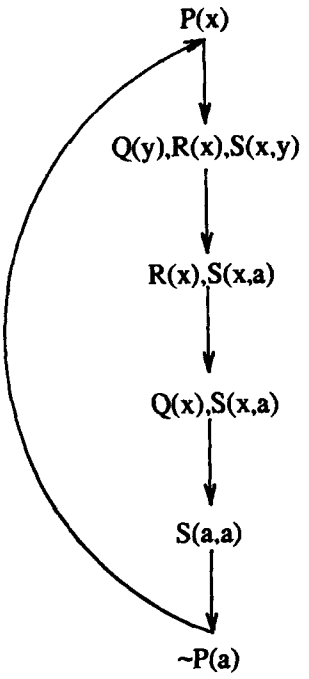


Figure A7. Prolog Derivation of ME1 Using ME Procedure.

The problem in the Prolog implementation that allows the set of clauses from Figure A8 to succeed occurs when Prolog attempts to unify lt(S(x), x) with lt(x, S(x)). It is obvious that there is no substitution ⊃ that will make x look like S(x), because x is contained in S(x). Therefore, any substitution for x will also be made in S(x). This is the "occur check" problem. If unification is allowed to proceed without making an occur check, then the unification will create an infinite data structure of the form x = S(S(S(S(. . . x . . .)))). The use of unification without the occur check is almost universal across Prolog

```
lt(x,
S(x))   ⊢

lt(3,2)   ⊢   lt(S(x),x)

          ?-  lt(3,2)
```

Figure A8. Illustration of the Dangers of Unification Without the Occur Check.

implementations. The motivation for using a unification algorithm that is unsound lies in the fact that the occur check can be very costly. For example, with a naive occur check, the concatenation of two lists requires (at least) a time proportional to the square of the size of the first list. If the occur check is eliminated, the time becomes linear (Colmerauer 1982).

One of the developers of Prolog, Alain Colmerauer, has suggested that it is actually a feature of the language that it allows the construction and manipulation of infinite data structures. He has constructed a formal semantics of Prolog using infinite trees (Colmerauer 1982); this semantics allows terms with loops since they may be considered to be infinite trees. In some situations, it may be useful to be able to have terms with loops; however, there are definite problems with this approach. First, it complicates the unification algorithm because it is possible to get into infinite recursion during the unification process, thus necessitating extra checks. Secondly, a semantics including infinite trees may not be familiar to most people; since there are many situations in which one intends the semantics to be that of FOPC. Finally, if the goal is to turn Prolog into a general theorem prover that is sound and complete for FOPC, one cannot accept the alternative semantics because this semantics is outside the framework of FOPC that is the target.

Several solutions to this problem have been proposed, involving various optimizations that would allow the occur check to be implemented with minimal impact to the computational efficiency. The remainder of this section focuses on two of those methods: the static worst-case analysis proposed by Plaisted, and the dynamic extension of the Warren virtual machine proposed by Beer. Since both of these methods pertain to a Prolog compiler rather than a Prolog interpreter, a briefly review and contrast of the two approaches will suffice.

Static Analysis

Plaisted proposed a procedure based upon static worst-case analysis of a Prolog program (Plaisted 1984). A set of instances of each clause that can be generated by an execution of the Prolog program is created. Mode declarations are provided for the top-level goal and a bipartite graph is then created that represents the set of "calling-literal/called-literal" pairs. Places where loops can be created are detected during the construction of the graph and appropriate tests are inserted at compile-time to prevent unsound unifications at run-time.

Dynamic Analysis

An alternative method was proposed by Joachim Beer that is based upon a fine-grained differentiation of the context in which the variables of a given clause occur (Beer 1988). It avoids the global analysis required by Plaisted's approach and involves only the analysis of variable occurrences local to a given clause. It is somewhat overly conservative, however, in comparison to the Plaisted approach in that it will indicate the necessity of including an occur check in a small percentage of cases where such a check is actually not necessary, Plaisted's approach will not make this error, so it is somewhat more

accurate. Beer's approach involves the extension of the virtual machine defined by Warren (1983). Prolog implementations use data tags or descriptors to identify the objects to be unified. Those data types are 'atom', 'integer', 'structure', etc.; however, unbound logical variables are generally only tagged as 'unbound variables'. No further information is provided as to the context in which the variable occurs. This is where the scheme presented by Beer extends Warren's virtual machine. Beer suggests the addition of abstract operations that place tags on variable reference. The first time a variable is encountered within a given clause, it is marked NEW_UNBOUND. If there is a subsequent reference to this variable within the clause, the tag is changed to UNBOUND. Whenever a structure or list is matched against a variable with the tag NEW_UNBOUND, no occur check needs to be executed, since NEW_UNBOUND indicates that there can be no other pointer to the variable and hence that no infinite loops can be created. However, if the tag is UNBOUND, the occur check will have to be done. Through the use of these tags to identify unbound variables and the context in which they occur, it is possible to avoid unnecessary occur checks.

Search Strategies

Depth-First Search

Even with the correction of the occurs check problem, Prolog's depth-first search strategy renders it unsatisfactory for theorem proving because it is incomplete. There are valid sets of unsatisfiable clauses for which Prolog will be unable to derive the empty clause because it will be forced to follow an infinitely deep path through the derivation space. As an example, consider the unsatisfiable set S1 shown in Figure A9. Whether or not a conventional Prolog program will actually find a proof is dependent upon the

$$\left\{ \begin{array}{l} Q(x) \supset P(x), S(x) \supset P(x), \\ P(x) \supset Q(x), S(a), \neg P(x) \end{array} \right\}$$

Figure A9. Sample Set S1.

ordering of the clauses in the input set. If the clauses are ordered as illustrated in Figure A10, then most Prologs will be sent into an infinite loop, expanding the cyclic relationship between P (x) and Q(x) forever. Such an infinite expansion may be seen by an examining the derivation space as illustrated in Figure A11. Prolog will descend the leftmost branch of the search tree, which is an infinitely deep path. In practice, this difficulty can often be avoided by a careful ordering of the input set, but there are some problems (such as attempting to axiomatize commutivity of a function), for which no such ordering will produce a well-behaved program. Even when such an ordering exists, it may not be obvious and it would be unacceptable for a general theorem prover to fail based on the order of the input set.

```
P(x)    ⊢    Q(x)
P(x)    ⊢    S(x)
Q(x)    ⊢    P(x)
S(a)    ⊢
        ?-   P(x)
```

Figure A10. Prolog Equivalent of Sample Set S1.

Bounded Depth-First Search

Obviously, to solve this problem, Prolog's unbounded depth-first search strategy must be replaced by a complete search strategy, i.e., one that is guaranteed to find an answer if one exists. Examples of complete search strategies are breadth-first search and the A algorithms. However, arbitrarily choosing a complete search strategy may result in the loss of much of the efficiency of Prolog implementations. In particular, adopting breadth-first search or the A algorithm would make it necessary for Prolog to represent and retain more than one derived clause at once, and such strategies would substantially increase memory requirements and reduce the efficiency of Prolog's deductive mechanism (Stickel 1985).

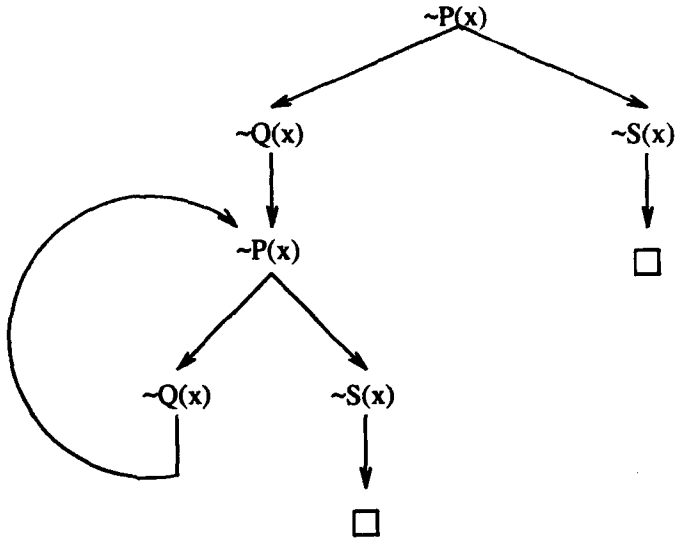


Figure A11. A Derivation Tree With an Infinite Path.

A simple solution to this problem is to replace Prolog's unbounded depth-first strategy with a bounded one. Backtracking when reaching the depth bound would cause the entire search space, up to a specified depth, to be searched completely. In the example search space of Figure A11, a depth-bound of 2 would cause Prolog to consider the entire search tree above and including the first recursive expansion.[2] This would result in recognition of the successful derivation path to 2 under ¬S(x) at depth 2 despite the fact that there exists an infinitely deep path along the left edge of the search tree that would have prevented a conventional implementation of Prolog from ever terminating the search.

In general, however, it is not possible to predict at what depth a successful derivation will be discovered. One way to avoid the problem of having to estimate the required depth that will be necessary to find a proof is to execute the bounded search with increasing depth-bounds. This is called depth-first iterative deepening (Korf 1985; Stickel and Tyson 1985). The effect is similar to breadth-first search except that results from earlier levels are recomputed rather than stored. Although this seems inefficient at first glance, the number of recomputed results is small in comparison with the size of the search space. With bounded depth-first search, it is also possible to augment the search with heuristics. For example, given an admissible heuristics that estimates the number of levels remaining to detect a proof, if the algorithm sees that the estimate to find a proof along the current derivational path exceeds the depth

[2]The root node is considered to be at depth 0.

bound, the path can be cut immediately. Further, if all such estimates are uniformly greater than the current depth-bound by a factor of N, then N levels may be skipped in establishing the next depth-bound during the interative-deepening process. As long as the estimate is admissible (i.e., never exceeds the actual number of remaining steps to a solution) then we are guaranteed of finding the shortest solution path first.

Example

This section contains an example that contrasts the behavior of conventional Prolog versus a Prolog that has been extended by the modifications suggested in the preceding sections.

Statement

The basic problem chosen was from page 93 of Genesereth's Logical Foundations of Artificial Intelligence (Genesereth and Nilsson 1988). A slightly modified form of the problem is presented here to facilitate the use of a single example to illustrate several points. First, the original problem will be stated and explained, and then the modifications will be addressed. The original problem statement was as follows:

Victor has been murdered, and Arthur, Bertram, and Carleton are suspects. Arthur says he did not do it. He says that Bertram was the victim's friend but that Carleton hated the victim. Bertram says he was out of town the day of the murder, and besides he didn't even know the guy. Carleton says he is innocent and he saw Arthur and Bertram with the victim just before the murder. Assuming that everyone except possibly for the murderer is telling the truth, use resolution to solve the crime.

There are several interesting aspects to this problem. First, not all the statements of the suspects may be accepted at face value, since it is made explicit that the murderer may not be telling the truth. Secondly, there are a number of unstated "common-sense" assumptions that may or may not need to be formalized to arrive at a resolution proof.

Because any one of the three suspects may be guilty, and the guilty suspect's testimony may not be trusted, this problem involves case-based reasoning. The statements of the three suspects must be represented in such a way as to capture the notion of conditional truth values. With a little consideration, it becomes obvious that the translation of the statements of the three suspects must be combined with a predicate indicating guilt or innocence in the formalization of the problem. Therefore, the truth of the suspect's statement is predicated on that suspect's innocence, and the guilty suspect may be found by finding the testimony that is contradictory to the other suspect's statements. There is an implicit assumption that only one of the three suspects is guilty; if this were not the case, the problem in its present form could not be solved. As it turns out, this insight is significant if we were attempting to reduce this problem to a set of Horn clauses to perform SLD resolution.

Common-sense knowledge enters into this problem as the attempt is made to formalize the implication of the suspect's testimony. An example of this is the two statements by Arthur and Bertram. Arthur maintains that Bertram was Victor's friend. Bertram, however, insists that he didn't know Victor. There is an implicit, common-sense relationship between knowing and being friends that must be captured in the axiomization of the problem. Which relationships are important to formalize is difficult to determine a priori; however, the performance of a knowledge-based system is tightly coupled with the ability to focus on the pertinent information and exclude extraneous, albeit possibly valid, relationships.

Problem Formalization

In the formalization of the problem, the following notation will be used:

A = Arthur
B = Bertram
C = Carleton
V = Victor
I(x) = x is innocent
P (x) = x is/was present
F (x, y)= x is/was a friend of y
L(x, y) = x likes/liked y
K(x, y) = x knows/knew y
W (x, y) = x is/was with y

The set of sentences derived from the problem statement is enumerated in the following paragraphs. Arthur said Bertram and Victor were friends, but that Carleton hated Victor:

$I(A) \supset F(B, V)$
$I(A) \supset \neg F(C, V)$

Bertram said that he was out of town the day of the murder and that he did not know Victor:

$I(B) \supset \neg P(B)$
$I(B) \supset \neg K(B, V)$

Carleton said that he saw both Arthur and Bertram with Victor just before the murder:

$I(C) \supset W(A, V)$
$I(C) \supset W(B, V)$

If someone was with Victor, then that person was present:

$\forall x\ W(x, V) \supset P(x)$

If person1 likes person2, or is person2's friend, then person1 must know person2:

$\forall x, y\ L(x, y) \supset K(x, y)$
$\forall x, y\ F(x, y) \supset K(x, y)$

Finally, it is necessary to make the assumption that two of the suspects are innocent (i.e., only one suspect is guilty). This is done by enumerating the possible combinations:

$\neg I(A) \supset I(B) \wedge I(C)$
$\neg I(B) \supset I(C) \wedge I(A)$
$\neg I(C) \supset I(A) \wedge I(B)$

Of course, when these statements are converted to clausal form, it may be seen that there is redundancy involved in this formulation. Specifically, these statements will reduce to three disjuncts of

two literals each, stating that A or B is innocent, A or C is innocent, and B or C is innocent. The conclusion we are seeking to prove is:

$$\exists x \neg I(x)$$

After analyzing the set of universal clauses used in the resolution derivation, it quickly becomes obvious that there is no set of Horn clauses that express these logical relationships. The first nine clauses are already in Horn clause format, but the last three clauses are disjuncts consisting of exclusively positive literals, and these cannot be represented by Horn clauses. A further problem exists: to be of use in an SLD derivation, each premise must be a Definite clause. Three of the nine premises already in Horn clause format are not Definite clauses, and therefore, could not contribute to a SLD derivation. Therefore, a conventional Prolog implementation would be unable to find a proof because the original problem statement lies outside Prolog's logical domain. Assuming a Prolog that has been extended with a true $\neg$ operator and that restriction concerning definite clauses has been relaxed, we can formulate the problem into a set of Prolog clauses, as illustrated in Figure A12. This results in the creation of A.24 clauses; since each of the original clauses consisted of two literals each, one extra contrapositive per clause would be created. The search tree for these clauses is illustrated in Figure A13. The "loops" in this diagram represent paths that generate nodes already contained in the path, thereby creating infinitely deep paths. The paths broken by two diagonal lines indicate that, in the interest of space, some intermediate nodes were omitted from the depiction of the search tree. It is interesting to note that there are infinitely deep paths along both edges of the search tree; therefore, a Prolog that was implemented with a conventional depth-first search strategy would again be unable to find the proof. Note also that a bounded search strategy is also not sufficient for finding the proofs in this instance; the proofs that do exist are found through use of the ME reduction operation. The first proof relies on the fact that $\neg W(B, V)$ has an ancestor $W(B, V)$. The second proof involves $I(C)$ and $\neg I(C)$. The third proof involves $I(A)$ and $\neg I(A)$. And finally, the last proof involves the goal $I(B)$ being a descendant of $\neg I(B)$. A Prolog implementation with bounded search but without the additional ME reduction inference operation would (in the worst case) still descend infinitely along four of the six initial paths, and would (in the best case) fail to find a proof, despite the fact that the set is unsatisfiable. This is hardly surprising, since a Prolog without some additional inference mechanism is attempting to apply input resolution in the unrestricted domain of FOPC, for which input resolution is known to be incomplete.

In comparing the performance of an extended Prolog theorem prover with a general theorem prover, such as FRAPPS4, it is interesting to note that the Prolog prover, given an initial depth-bound of 8 (which is optimal for this problem), would have generated 27 nodes before finding a solution. Despite the use of deletion strategies to constrain the size of the search space, FRAPPS still generated 131 clauses, as can be seen from the fragment of FRAPPS output included in Figure A14. Therefore, the extended-Prolog strategy of using an extension of input resolution in conjunction with a selection function results in a marked increase in the efficiency of the navigation of the search space.

Framework for Resolution-based Automated Proof Procedure System (FRAPPS) was developed by Prof Alan Frisch, Michael K. Mitchell, and others at the University of Illinois. It consists of a package of Common Lisp functions that provide a set of building blocks from which a resolution-based deduction system can be easily constructed.

F(B,V)	⊢	I(A)	(A.1)
I(A)	⊢	¬I(B)	(A.2)
I(A)	⊢	¬I(C)	(A.3)
I(B)	⊢	¬I(A)	(A.4)
I(B)	⊢	¬I(C)	(A.5)
I(C)	⊢	¬I(A)	(A.6)
I(C)	⊢	¬I(B)	(A.7)
K(x,y)	⊢	L(x,y)	(A.8)
K(x,y)	⊢	F(x,y)	(A.9)
P(x)	⊢	W(x,V)	(A.10)
W(A,V)	⊢	I(C)	(A.11)
W(B,V)	⊢	I(C)	(A.12)
¬F(x,y)	⊢	¬K(x,y)	(A.13)
¬I(A)	⊢	¬F(B,V)	(A.14)
¬I(A)	⊢	L(C,V)	(A.15)
¬I(B)	⊢	P(B)	(A.16)
¬I(B)	⊢	K(B,V)	(A.17)
¬I(C)	⊢	¬W(A,V)	(A.18)
¬I(C)	⊢	¬W(B,V)	(A.19)
¬K(B,V)	⊢	I(B)	(A.20)
¬L(B,V)	⊢	I(A)	(A.21)
¬L(x,y)	⊢	¬K(x,y)	(A.22)
¬P(B)	⊢	I(B)	(A.23)
¬W(x,V)	⊢	¬P(x)	(A.24)

Figure A12. Prolog Clause Set.

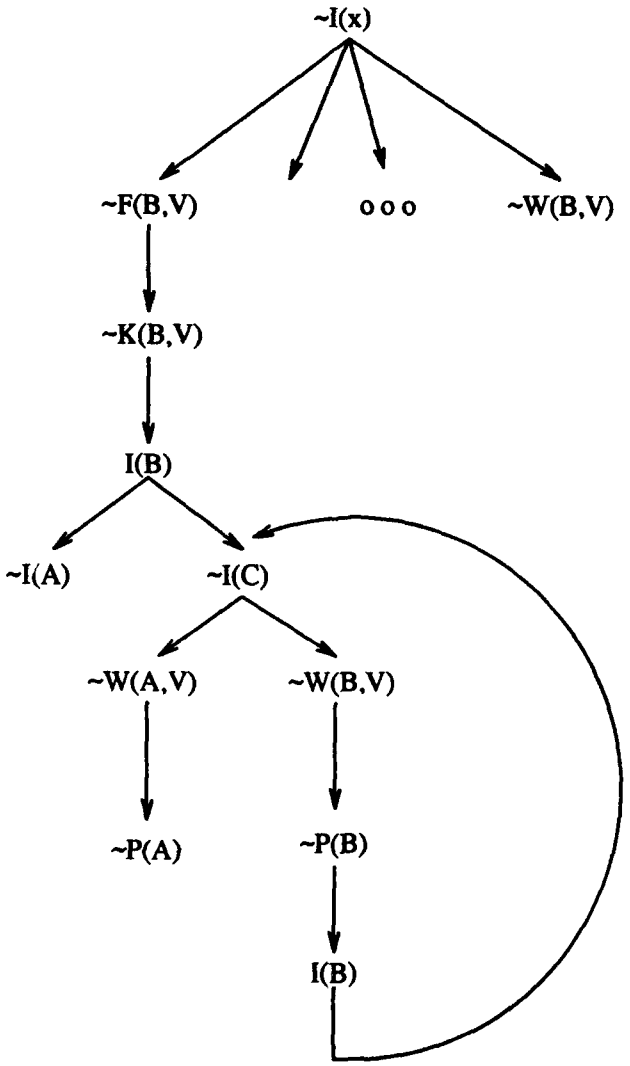


Figure A13. Search Tree in Problem Reduction Format.

```
- 1: ((NOT I A) (F B V))
- 13: ((I (*VAR* X 1)))
- 8: ((NOT F (*VAR* X 1) (*VAR* Y 1))
              (K (*VAR* X 1) (*VAR* Y 1)))
------- 14: ((F B V))(1 13)
- 4: ((NOT I B) (NOT K B V))
- 13: ((I (*VAR* X 1)))
------- 17: ((NOT K B V)) (4 13)
---------- 129: ((K B V)) (8 14)
---------- 131: NIL (17 129)
NIL
```

Figure A14. Solution Using General Resolution.

APPENDIX B: ModLog Syntax

The following chapter describe the syntax of legal expressions in the ModLog language.

input	⇒	clause
	∣	query
clause	⇒	'IF *goallist* 'THEN' goal '.'
	∣	*goal* '.'
query	⇒	*goallist* '?'
goallist	⇒	*goal* [*connector goal*]
goal	⇒	*name*
	∣	*name* '(' *expresslist* ')'
	∣	*variable*
	∣	*variable* ':=' *simplexpr*
expresslist	⇒	*expression* [*connector expression*]
expression	⇒	*integer*
	∣	*real*
	∣	*char*
	∣	*string*
	∣	*variable*
	∣	*functor*
	∣	*mtl*
	∣	*conslist*
functor	⇒	*name*
	∣	*name* '(' *expresslist* ')'
conslist	⇒	'[' *consexprlist* ']'
	∣	'[' *consexprlist* '|' *variable* ']'
consexprlist	⇒	*expression* [',' *expression*]
simexprlist	⇒	*simplexpr* [*connector simplexpr*]
simplexpr	⇒	*term* [*addop term*]1
addop	⇒	'+'
	∣	'-'
term	⇒	*factor* [*mulop factor*]1
mulop	⇒	'*'
	∣	'/'
	∣	'^'
	∣	'%'
factor	⇒	*integer*
	∣	*real*
	∣	*variable*
	∣	'(' *simplexpr* ')'
	∣	*name* '(' *simexprlist* ')'
connector	⇒	'*and*'
	∣	','
name	⇒	[a–z][a–z0–9]
variable	⇒	[A–Z][a–z0–9]
integer	⇒	[0 9]
	∣	'(' '-' [0–9] ')'
real	⇒	[0–9]+ '.' [0–9] ['E' ['+' | '-']1 [0–9]+]1 ∣ '(' '-' [0–9]+ '.' [0–9] ['E' ['+' ! '-']1 [0–9]+]1 "'

char	⇒	‘’’ [a zA Z0 9:::]1 ‘’’
string	⇒	‘”’ *char* ‘”’
mtl	⇒	‘[]’

APPENDIX C: DOME Syntax

knowmod	⇒	KNOWLEDGE MODULE *name* ;[*objsect*]+ END MODULE.
objsect	⇒	OBJECT *name* ;[*clause*]+ END OBJECT.
clause	⇒	*goallist* THEN *goal* .
	❙	*goal* .
goallist	⇒	*goal* [*connector goal*]
goal	⇒	[IF]$^{?}$ *name* [(*elist*)]$^{?}$ [IS TRUE]$^{?}$
	❙	[IF]$^{?}$ *variable* [:= (*sexpr*)]$^{?}$ [IS TRUE]$^{?}$
elist	⇒	*expr* [*connector expr*]
expr	⇒	*integer*
	❙	*real*
	❙	*char*
	❙	*string*
	❙	*variable*
	❙	*functor*
	❙	*mtlist*
	❙	*conslist*
functor	⇒	*name* [(*elist*)]$^{?}$
conslist	⇒	[*celist* [— *variable*]$^{?}$]
celist	⇒	*expr* [, *expr*]
mtlist	⇒	[]
	❙	NIL
variable	⇒	*<Any string of characters starting with a '?'>*
name	⇒	*<Any string of characters starting with a alphanumeric.>*
connector	⇒	AND
	❙ ,	[AND]$^{?}$
integer	⇒	[0 .. 9]+
real	⇒	[0 .. 9]+ .[0 .. 9]+
char	⇒	' *<Any character>* '
string	⇒	"[*<Any character except '"'>*] "

APPENDIX D: Primitive Predicates

This chapter describe the primitive predicates that are available in ModLog. As a general rule, whenever a predicate allows arguments, one (and only one) of the arguments may be a variable. In this case, the predicate will attempt to instantiate the variable with a suitable value. If the predicate is unable to do so, it will fail (i.e., return FALSE). If none of the arguments are variables, the predicate will evaluate the given arguments to see if they satisfy the relationship specified by the predicate. Most predicates require a specific number of arguments. Those that allow a variable number of arguments are annotated with the ellipse (i.e., ...) in their argument lists.

Arithmetic Predicates

The following predicates handle arithmetic operations. These predicates handle arguments that are either integer or real values; however, unless otherwise noted, the data types cannot be mixed (i.e., all the arguments must be the same data type or a variable integers and reals cannot occur in the same predicate).

divide(term1, term2, term3)

This predicate returns TRUE if *term3* is the result of dividing *term1* by *term2*. It works for either integers or reals (but the data types cannot be mixed).

minus(term1, term2, term3)

This predicate returns TRUE if *term3* is the result of subtracting *term2* from *term1*. It works for either integers or reals (but the data types cannot be mixed).

plus(term1, term2, term3)

This predicate returns TRUE if *term3* is the result of adding *term1* to *term2*. It works for either integers or reals (but the data types cannot be mixed).

times(term1, term2, term3)

This predicate returns TRUE if *term3* is the result of multiplying *term1* by *term2*. It works for either integers or reals (but the data types cannot be mixed).

sqrt(term1, term2)

This predicate returns TRUE if *term2* is the square root of *term1*. If *term1* is a variable, it is bound to the square of *term2*. If *term1* is an integer, then sqrt fails if there is no perfect square root (i.e., it will not return a real result from an integer argument).

Arithmetic Assignment

There is a special form of expression recognized by ModLog, referred to as arithmetic assignment. An expression of the form

V ar := expr

signals an arithmetic assignment. In this expression, V ar must be and unbound variable, and expr must be an arithmetic expression consisting of some combination of the following operators: +, , , =, ^, %, *acos(x)*, *asin(x)*, *atan(x)*, *ceil(x)*, *cos(x)*, *cosh(x)*, *exp(x)*, *fabs(x)*, *float(x)*, *floor(x)*, *log10(x)*, *log(x)*, *sin(x)*, *inh(x)*, *sqrt(x)*, *tan(x)*, *tanh(x)*, *trunc(x)*, *atan2(x)*, *fmod(x; y)*, and *pow(x; y)*. Arithmetic expression must be balanced. In other words,

X := (a + b) + c

is legal, while

X := a + b + c

is not, because the latter expression is ambiguous (i.e., can be understood two ways: as ((a +) + c) and as (a + (b + c)). When in doubt, parenthesize. All of the following functional operators returns real results, except for trunc.

acos(x)

Returns the arccosine of its argument (in radians).

asin(x)

Returns the arcsine of its argument (in radians).

atan(x)

Returns the arctangent of its argument (in radians).

ceil(x)

Returns the closest integral number greater than x.

cos(x)

Returns the cosine of its argument (an angle in radians).

cosh(x)

Returns the hyperbolic cosine of its argument.

exp(x)

Returns e raised to the x power.

fabs(x)

Returns the absolute value of x.

float(x)

Returns the floating point value equal to x.

floor(x)

Returns the closest integral values less than x.

log10(x)

Returns the log base 10 of x.

log(x)

Returns the log base e of x.

sin(x)

Returns the sine of x.

sinh(x)

Returns the hyperbolic sine of x.

sqrt(x)

Returns the square root of x.

tan(x)

Returns the tangent of x.

tanh(x)

Returns the hyperbolic tangent of x.

trunc(x)

Returns the integer value equal to x.

atan2(x,y)

Returns the arctangent of x=y.

fmod(x,y)

Returns x mod y.

pow(x,y)

Returns x raised to the y power.

Interactive Predicates

The following predicates would only be used during development, as their only function is to provide information about the state of the database through printing that information to the terminal screen. For this reason, they are not present in the embedded version of ModLog.

listing

Produces a listing of the clauses in the database. The ordering of clauses reflects the order that the clauses occur in the database (i.e., their order is preserved). This predicate always returns TRUE.

prims

Lists the primitive symbols currently registered in the interpreter. This predicate always returns TRUE.

trace

This predicate toggles tracing on and off. As clauses are evaluated and goals expanded, information about the current goal state is dumped to the terminal screen. This predicate always returns TRUE.

print(term1, term2, ...)

Comparison Predicates

The following predicates compare their arguments.

compare(term1, term2, term3)

This is the most general comparison predicate. It takes three arguments, compares *term1* to *term2* and binds *term3* to the integer result of the comparison. If the two arguments are equal, *term3* is bound to 0. If *term1* is less than *term2*, then *term3* is bound to an integer less than 0. If *term1* is greater than *term2*, then *term3* is bound to an integer greater than zero. For a precise definition of equality, see the description under the predicate equal. For a more detailed description of inequality, see the discussion under the predicate greater.

equal(term1, term2)

This predicate returns TRUE if *term1* is equal to *term2*. If two terms of differing data types are compared, these terms are unequal by definition. Two unbound variables are also equal by definition. Integers and Reals are compared numerically. Characters and Strings are compared lexigraphically. Functor expressions are compared first by length, then lexigraphically by functor, and finally recursively by arguments.

greater(term1, term2)

This predicate returns TRUE if *term1* is greater than *term2*. By definition, differing data types are ranked in the following order (from least to greatest): variables, integers, reals, characters, strings, functor expressions. Therefore, if a character is compared to an integer, it is always greater than the integer, regardless of the actual value of the character and integer in question. If two unbound variables are

compared, this predicate returns TRUE; otherwise, see the discussion under the predicate equal for a discussion of comparing other data types.

greatereq(term1, term2)

This predicate returns TRUE if *term1* is greater than or equal to *term2*. See the discussion under greater for more details.

less(term1, term2)

This predicate returns TRUE if *term1* is greater than or equal to *term2*. See the discussion under greater for more details.

lesseq(term1, term2)

This predicate returns TRUE if *term1* is greater than or equal to *term2*. See the discussion under greater for more details.

unequal(term1, term2)

This predicate returns TRUE if *term1* is greater than or equal to *term2*. See the discussion under equal for more details.

Control Predicates

The following predicates interact with the database of clauses.

assert(term1, term2, ...)

This predicate asserts the terms that are its arguments (which must be functor expressions) into the database following the last term with the same functor.

asserta(term1, term2, ...)

This predicate asserts the terms that are its arguments (which must be functor expressions) into the database before the first term with the same functor.

clause(term1, term2)

This predicate unifies *term1* (which must be a functor expression) with the database and binds *term2* (which must be a variable) to one of the list of clauses in the body of *term1*. This predicate is backtracking. If *term1* can be found in the database, this predicate returns TRUE. If *term1* has no body, *term2* is bound to the empty list (i.e., []).

fail

This predicate always returns FALSE. It is useful to force iterative evaluation of a set of clauses.

not(term1, term2, ...)

This predicate attempts to prove the terms that are its arguments and returns FALSE if it succeeds in doing so. If it cannot prove the conjunction of the terms, it returns TRUE.

or(term1, term2, ...)

This predicate attempts to complete the proof with each of its arguments in turn, succeeding on the first one that succeeds. It fails if the proof cannot be completed with any of its terms.

save("filename")

The entire contents of the database will be written to the file "filename". If the file already exists, its contents will be overwritten. This predicate always returns TRUE.

see("filename")

This predicate opens a file named by filename (a fully instantiated string) and reads the clauses in the file into the database. These clauses must be syntactically correct according to ModSim syntax. If the named file exists, see will attempt to read and act on the entire contents of the file, and then return TRUE. If the file does not exist, see returns FALSE.

retract(term)

This predicate removes from the database the first clause that unifies with term. If no clause in the database unifies with term, then retract returns FALSE; otherwise, TRUE is returned.

retractall(term)

This predicate removes from the database all clauses that unify with term. This predicate always returns TRUE.

unify(term1, term2)

This predicate returns TRUE if *term1* unifies with *term2*.

Conversion Predicates

The following predicates provide the ability to convert between data types.

str2atom(term1, term2)

This predicate converts strings to atomic values, or atomic values to strings. Upon successful completion, *term1* will be a string and *term2* will be an atomic value (integer, real, char, or string).

str2char(term1, term2)

This predicate converts strings to character values, or character values to strings. Upon successful completion, *term1* will be a string and *term2* will be the first character in that string.

str2int(term1, term2)

This predicate converts strings to integer values, or integer values to strings. Upon successful completion, *term1* will be a string and *term2* will an integer.

str2list(term1, term2)

This predicate converts strings to lists of character values, or lists of character values to strings. Upon successful completion, *term1* will be a string and *term2* will be a list containing the characters in that string.

str2real(term1, term2)

This predicate converts strings to real values, or real values to strings. Upon successful completion, *term1* will be a string and *term2* will a real.

str2sym(term1, term2)

This predicate converts strings to symbolic (functor) values, or symbolic values to strings. Upon successful completion, *term1* will be a string and term2 will a symbol.

univ(term1, term2)

This predicate converts a functor expression into a list, or a list into a functor expression. *term1* must either be a functor expression or a variable, while *term2* must either be a list or a variable. It returns TRUE if the conversion is possible.

Counter Predicates

The following predicates operate on each ModLog object's internal counters. There are 10 counters in all, numbered 0 through 9.

cntval(term1, term2)

This predicate returns TRUE if the value of counter(*term1*) equals *term2*. If *term1* is a variable, then *term1* is unified with the first counter whose value is the integer *term2*.

cntset(term1, term2)

This predicate sets the value of counter(*term1* to the integer represented by *term2*. If *term2* is a variable, it is unified with the value of the counter, and the counter's value remains unchanged. If *term1* is a variable, then *term1* is unified with the first counter whose value is equal to *term2*.

cntinc(term1, term2)

This predicate increments the value of counter(*term1* by one and returns TRUE if that value unifies with *term2*. If *term1* is a variable, then *term1* is unified with the first counter whose incremented value is equal to *term2*.

cntdec(term1, term2)

This predicate decrements the value of counter(*term1* by one and returns TRUE if that value unifies with *term2*. If *term1* is a variable, then *term1* is unified with the first counter whose decremented value is equal to *term2*.

Data Type Predicates

The following predicates test their arguments to see if they are of a particular data type (or set of data types).

intp(term)

This predicate returns TRUE if term is a ModLog integer; otherwise, it returns FALSE.

realp(term)

This predicate returns TRUE if term is a ModLog real number; otherwise, it returns FALSE.

charp(term)

This predicate returns TRUE if term is a ModLog character; otherwise, it returns FALSE.

stringp(term)

This predicate returns TRUE if term is a ModLog string; otherwise, it returns FALSE.

unboundp(term)

This predicate returns TRUE if term is a ModLog unbound variable; otherwise, it returns FALSE.

numberp(term)

This predicate returns TRUE if term is a ModLog integer or real; otherwise, it returns FALSE.

symbolp(term)

This predicate returns TRUE if term is a ModLog symbol; otherwise, it returns FALSE.

listp(term)

This predicate returns TRUE if term is a ModLog list; otherwise, it returns FALSE.

atomp(term)

This predicate returns TRUE if term is a ModLog atomic data type (i.e., an integer, real, character, string, or symbol); otherwise, it returns FALSE.

List Predicates

The following predicates manipulate "cons-lists."

listlen(term1, term2)

This predicate operates on lists. *term1* must either be a list or an unbound variable. If *term1* is a list, length computes the length of the list and attempts to unify that integer with *term2*, returning TRUE if successful. If *term1* is a variable, then *term2* must be an integer and a list of length *term2* is manufactured.

member(term1, term2)

This predicate operates on lists, returning TRUE if *term1* is a member of the list *term2*. If *term1* is a variable, *term2* must be an instantiated list, and member will backtrack through each member of the list. If *term2* is a variable, a list will be constructed that has *term1* as its only element. If both *term1* and *term2* are instantiated, then the list is searched for an occurrence of *term1*.

nelem(term1, term2, term3)

This predicate operates on lists. It returns TRUE if *term3* is the *term2*th element of list *term1* (counting from 1). For example, `nelem([a,b,c],2,b)` would return TRUE.

String Predicates

The following predicates operate on strings.

concat(term1, term2, term3)

This predicate concatenates strings. It returns TRUE if *term3* is the concatenation of *term1* and *term2*. Any one of the terms can be a variable, in which case concat attempts to satisfy the relationship by constructing the appropriate sub-string.

stridx(term1, term2, term3)

This predicate selects characters from a string. It returns TRUE if *term3* is the character at the *term2*th position of string *term1* (counting from 1).

strlen(term1, term2)

This predicate returns TRUE if *term2* unifies with the integer corresponding to the length of the string that unifies with *term1*; otherwise, it returns FALSE. If *term1* is a variable and *term2* and integer, an arbitrary string of length *term2* will be created.

DISTRIBUTION

Chief of Engineers
ATTN: CEHEC-IM-LH (2)
ATTN: CEHEC-IM-LP (2)
ATTN: CERD-L
ATTN: DAEN-ZCM

Fort Belvoir, VA 22060-5516
ATTN: CECC-R

Defense Technical Info. Center 22304
ATTN: DTIC-FAB (2)

9
+84
9/93

This publication was reproduced on recycled paper.

☆ U.S. GOVERNMENT PRINTING OFFICE: 1993—3510-S/80060